More Examples of Creating Loops

1. Create a program for binary search $BS(b, x)$ that searches for x (where $b[0] \le x \le b[n-1]$) in sorted array b (with no duplicated numbers) whose $size(b) = n > 0$. In the postcondition, let L be the index of x if and only if x is in b, $0 \le L < n$.

   - How does binary search work? $[L, R)$

     We let L and R be the left (inclusive) and right (exclusive) boundary of the "search range": elements in this range are candidate for target x. We know that the idea of binary search is to "halve" the search range after each iteration (by letting mid-point m be the new L or R in the next iteration) until the target x is found or the search range is shrunk to size 0 (or 1). We need to be careful while guaranteeing that the search range shrinks after each iteration: because when close to the end of the loop, it is possible that $R = L + 1$, then $m = \frac{L+R}{2} = L$ which might lead to divergence. Here let's look one way to solve this problem (we terminate the loop when the search range has size 1):

     We assign $L \coloneqq m$ or $R \coloneqq m$ in each iteration, then we need to terminate with $\underline{R = L + 1}$ (which means there is only $b[L]$ in the search range) if x in not found. Because R can be $L + 1$ and L can be $n - 1$, we need to artificially define $b[n] = b[n-1] + 1$.

     (As an aside, another design can be let $L \coloneqq m + 1$ or $R \coloneqq m$ after each iteration. Think about how to create a loop using this design.)

     - The precondition of the program can be $Sorted(b) \land size(b) = n > 0 \land b[0] \le x \le b[n-1]$, where $Sorted(b) \equiv \forall 0 \le k < size(b) - 1.\, b[k] < b[k+1]$. Since $Sorted(b)$ is always true during the program and our searching procedure doesn't change it; we will only show it in the precondition and omit it everywhere else.

     - In the postcondition, to show whether we find x, we introduce a Boolean variable $found$. We say $found$ only if we have $b[L] = x$ at the end. The postcondition of the program can be written as $q \equiv (0 \le L < n) \land (b[L] \le x < b[L+1]) \land (found \rightarrow b[L] = x)$.

     - For loop invariant:
       - Dropping off either conjunct doesn't look promising. (The dropped conjunct will be $\neg B$.)
       - Replacing $L + 1$ by a variable R can be a good idea, and we have $R \ne L + 1$ while looping. The range of R should be $L + 1 \le R \le n$, and we can end the loop with either $R = L + 1$ or $found$.

       - In the end, we can try to use loop invariant $p \equiv (0 \le L < R \le n) \land (b[L] \le x < b[R]) \land (found \rightarrow b[L] = x)$
       - At the same time, we find that we should try to use loop condition $B \equiv \neg found \land R \ne L + 1$.

     - For the bound expression: we will increase L or decrease R in each iteration, and loop invariant implies $R > L$, so we can use $R - L$ as the bound expression.

     - Then we can come up with the following partial program:

       $\{Sorted(b) \land size(b) = n > 0 \land b[0] \le x \le b[n-1] < b[n]\}$

…

$\{\mathbf{inv}\ p \equiv (0 \le L < R \le n) \wedge (b[L] \le x < b[R]) \wedge (found \to b[L] = x)\}$ $\{\mathbf{bd}\ R - L\}$

$\mathbf{while}\ \neg found \wedge R \ne L + 1\ \mathbf{do}$

$\{p \wedge \neg found \wedge R \ne L + 1 \wedge R - L = t_0\}$

$m := (L + R) \div 2;$

$\{p_1 \equiv p \wedge \neg found \wedge R \ne L + 1 \wedge R - L = t_0 \wedge m = (L + R) \div 2\}$ # forward assignment

$\mathbf{if}\ b[m] = x\ \mathbf{then}$

$\{p_1 \wedge b[m] = x\}\ found := T; L := m\ \{p \wedge R - L < t_0\}$

else

$\{p_1 \wedge b[m] \ne x\}$ … $\{p \wedge R - L < t_0\}$

fi

$\{p \wedge R - L < t_0\}$

od

$\{0 \le L < R \le n \wedge (b[L] \le x < b[R]) \wedge (found \to b[L] = x) \wedge (found \vee R = L + 1)\}$

$\{q \equiv 0 \le L < n \wedge (b[L] \le x < b[L + 1]) \wedge (found \to b[L] = x)\}$

- There are still gaps in the program and we don't have a full proof outline yet.
  - To get $\mathbf{inv}\ p$ from the precondition, we need to initialize values of L, R and $found$.
  - The true branch lacks proof: so, we need to add forward or backward assignments between statements. We omit the proof here.

  - For the false branch: we can use another conditional statement to assign $L := m$ or $R := m$. Then we have the following partial proof outline under total correctness:

$\{Sorted(b) \wedge 1 \le n = size(b) \wedge b[0] \le x \le b[n - 1] < b[n]\}$

$L := 0; R := n; found := F;$

$\{1 \le n = size(b) \wedge b[0] \le x \le b[n - 1] < b[n] \wedge L = 0 \wedge R = n \wedge found = F\}$

$\{\mathbf{inv}\ p \equiv 0 \le L < R \le n \wedge (b[L] \le x < b[R]) \wedge (found \to b[L] = x)\}$ $\{\mathbf{bd}\ R - L\}$

$\mathbf{while}\ \neg found \wedge R \ne L + 1\ \mathbf{do}$

$\{p \wedge \neg found \wedge R \ne L + 1 \wedge R - L = t_0\}$

$m := (L + R) \div 2;$

$\{p_1 \equiv p \wedge \neg found \wedge R \ne L + 1 \wedge R - L = t_0 \wedge m - (L + R) \div 2\}$

$\mathbf{if}\ b[m] = x\ \mathbf{then}$

$\{p_1 \wedge b[m] = x\}\ found := T; L := m; \{p \wedge R - L < t_0\}$

else

$\{p_1 \wedge b[m] \ne x\}\ \mathbf{if}\ b[m] > x\ \mathbf{then}\ R := m\ \mathbf{else}\ L := m\ \mathbf{fi}\ \{p \wedge R - L < t_0\}$

fi

$\{p \wedge R - L < t_0\}$

od

$\{p \wedge (found \vee R = L + 1)\}$

$\{q \equiv 0 \le L < n \wedge (b[L] \le x < b[L + 1]) \wedge (found \to b[L] = x)\}$

2. Given two non-empty sorted arrays b_1 and b_2, find the least indices i and j such that $b_1[i] = b_2[j]$; if no such i and j exist, end with $i = n$ or $j = m$ such that $n = size(b_1)$ and $m = size(b_2)$.

   - We have seen the three-array version of this problem when we introduce nondeterministic statements: our algorithm starts with $i = j = 0$ then increase either i or j in each iteration. This time we focus on the loop invariant and termination.

- The precondition of the program can be: $size(b_1) = n > 0 \wedge size(b_2) = m > 0 \wedge Sorted(b_1) \wedge Sorted(b_2)$. Since $Sorted(b_1) \wedge Sorted(b_2)$ is always true during the program and our searching procedure doesn't change it; we will only show it in the precondition and omit it everywhere else.

- While discussing the three-array version, we mentioned that our algorithm will only return the left-most match. So, if the program ends with $b_1[i]$ and $b_2[j]$, then there is no match on the left of i and j. This is similar to the postcondition of the linear search, we can write postcondition $q \equiv 0 \le i \le n \wedge 0 \le j \le m \wedge NoMatch(b_1, b_2, i, j) \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$, where $NoMatch(b_1, b_2, i, j) \equiv \forall 0 \le i' < i. \forall 0 \le j' < j. b_1[i'] \ne b_2[j']$.

- Like linear search, we can also get invariant by dropping of the last conjunct, and $p \equiv 0 \le i \le n \wedge 0 \le j \le m \wedge NoMatch(b_1, b_2, i, j)$, then we have loop condition $B \equiv \neg(i < n \wedge j < m \rightarrow b_1[i] = b_2[j]) \Leftrightarrow i < n \wedge j < m \wedge b_1[i] \ne b_2[j]$.

- Since we are increase i and j, so we define bound function $t(i,j) \equiv (n - i) + (m - j)$. It is easy to see that $t(i,j) \ge 0$.
  Then we can get the following partial program:

  $\{size(b_1) = n > 0 \wedge size(b_2) = m > 0 \wedge Sorted(b_1) \wedge Sorted(b_2)\}$
  $\{0 \le 0 \le n \wedge 0 \le 0 \le m \wedge NoMatch(b_1, b_2, 0, 0)\}$
  $i := 0; j := 0;$
  $\{\mathbf{inv}\ p \equiv 0 \le i \le n \wedge 0 \le j \le m \wedge NoMatch(b_1, b_2, i, j)\}\ \{\mathbf{bd}\ t(i,j) \equiv (n - i) + (m - j)\}$
  $\mathbf{while}\ B \equiv i < n \wedge j < m \wedge b_1[i] \ne b_2[j]\ \mathbf{do}$
  $\qquad \{p \wedge B \wedge t(i,j) = t_0\}$
  $\qquad \ldots$ increase i or j, and maybe something else …
  $\qquad \{p \wedge t(i,j) < t_0\}$
  $\mathbf{od}$
  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}$
  $\{q \equiv 0 \le i \le n \wedge 0 \le j \le m \wedge NoMatch(b_1, b_2, i, j) \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}$

- What program should be the loop body? Let's use a deterministic program this time. We want to increase $i := i + 1$ when $b_1[i] < b_2[j]$, since $b_2[j]$ is already too large so increasing j won't help; symmetrically, we want to increase $j := j + 1$ when $b_1[i] > b_2[j]$. With a conditional statement, we have the following partial proof outline:

  $\{size(b_1) = n > 0 \wedge size(b_2) = m > 0 \wedge Sorted(b_1) \wedge Sorted(b_2)\}$
  $\{0 \le 0 \le n \wedge 0 \le 0 \le m \wedge NoMatch(b_1, b_2, 0, 0)\}$
  $i := 0; j := 0;$
  $\{\mathbf{inv}\ p \equiv 0 \le i \le n \wedge 0 \le j \le m \wedge NoMatch(b_1, b_2, i, j)\}\ \{\mathbf{bd}\ t(i,j) \equiv (n - i) + (m - j)\}$
  $\mathbf{while}\ B \equiv i < n \wedge j < m \wedge b_1[i] \ne b_2[j]\ \mathbf{do}$
  $\qquad \{p \wedge B \wedge t(i,j) = t_0\}$
  $\qquad \mathbf{if}\ b_1[i] < b_2[j]\ \mathbf{then}$ # Conditional Rule 1
  $\qquad\qquad \{p \wedge B \wedge t(i,j) = t_0 \wedge b_1[i] < b_2[j]\}\ i := i + 1\ \{p \wedge t(i,j) < t_0\}$
  $\qquad \mathbf{else}$ $\# b_1[i] > b_2[j]$
  $\qquad\qquad \{p \wedge B \wedge t(i,j) = t_0 \wedge b_1[i] > b_2[j]\}\ j := j + 1\ \{p \wedge t(i,j) < t_0\}$
  $\qquad \mathbf{fi}$
  $\qquad \{p \wedge t(i,j) < t_0\}$
  $\mathbf{od}$
  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}$
  $\{q \equiv 0 \le i \le n \wedge 0 \le j \le m \wedge NoMatch(b_1, b_2, i, j) \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}$

3. Create a program that can find some x such that $x \le sqrt(n) < x + 1$, where $n \ge 0$ is given in the question; or equivalently, $x^2 \le n < (x+1)^2$.

1) The postcondition here is straightforward: $x^2 \le n < (x+1)^2$.
   The precondition can simply be: $n \ge 0$

2) How about replacing the "2" in the power with a variable k?
   If $p \equiv x^k \le n < (x+1)^2$, then the loop will end up with $k = 2$. What should be the other boundary of the range of k? If the other boundary is larger than 2, then we probably won't be able to find any k such that $x^k < (x+1)^2$; if it is smaller than 2, then it is either 0 or 1, then this loop is trivial since there are only three possible values of k and I don't see it is helpful for looking for x. So, this is not a good idea.

   Replacing $(x+1)^2$ with $(x+1)^k$ also doesn't help us much about looking for x and we omit the discussion here.

3) Replacing the constant 1 with k can be a good idea for loop invariant. The loop ends up with $k = 1$, and $k = n + 1$ can be large enough as an upper bound for k, so k can have range $1 \le k \le n + 1$.

   In each iteration, to find such x, we need to either make x larger or make k smaller, this implies that $-x + k + n$ is a good bound expression.

   $\{n \ge 0\}$ ...
   $\{\mathbf{inv}\ x^2 \le n < (x+k)^2 \land 1 \le k \le n+1\}\{\mathbf{bd}\ -x+k+n\}$
   while $k \ne 1$ **do**
       ... increase x or decrease k, and maybe something else ...
   od
   $\{x^2 \le n < (x+k)^2 \land 1 \le k \le n+1 \land k = 1\}$ $\# p \land \neg B$
   $\{x^2 \le n < (x+1)^2\}$

4) We can try to use the idea of binary search to shrink the range: In each iteration we can compare n with the middle point $(x + k \div 2)^2$ to decide whether we want to increase the lower bound or the upper bound of the searching range. In addition, to add a precondition, we can start the search with $x = 0$ and $k = n + 1$. We can get the following partial proof outline.

   $\{n \ge 0\}\ x := 0; k := n+1; \{n \ge 0 \land x = 0 \land k = n+1\}$
   $\{\mathbf{inv}\ p \equiv x^2 \le n < (x+k)^2 \land 1 \le k \le n+1\}\{\mathbf{bd}\ -x+k+n\}$
   while $k \ne 1$ **do**
       $\{p \land k \ne 1 \land -x+k+n = t_0\}$
       if $(x + k \div 2)^2 > n$ **then**
           $k := k \div 2$
       else $\#(x + k \div 2)^2 \le n$
           $x := x + k \div 2; k := k - k \div 2$
       $\{p \land -x+k+n < t_0\}$
   od
   $\{x^2 \le n < (x+k)^2 \land 1 \le k \le n+1 \land k = 1\}$
   $\{x^2 \le n < (x+1)^2\}$

5) As an aside, we can see that k is decreasing in every iteration, we can simply use k instead of $-x + k + n$ as the bound expression.